Encrypting data with the Blowfish algorithm

Bill Gatliff (/user/Bill Gatliff)

JULY 15, 2003

Many embedded systems depend on obscurity to achieve security. We often design systems to download unsigned or unencrypted firmware upgrades or store unencrypted user data, a practice we justify because it's invisible to the end user and makes our lives easier. The stealthy practice, however, is no longer kosher. With the help of this public-domain encryption algorithm, we can clean up our act.

Modern embedded systems need data security more than ever before. Our PDAs store personal e-mail and contact lists; GPS receivers and, soon, cell phones keep logs of our movements;[1] and our automobiles record our driving habits.[2] On top of that, users demand products that can be reprogrammed during normal use, enabling them to eliminate bugs and add new features as firmware upgrades become available.

Data security helps keep private data private. Secure data transmissions prevent contact lists and personal e-mail from being read by someone other than the intended recipient, keep firmware upgrades out of devices they don't belong in, and verify that the sender of a piece of information is who he says he is. The sensibility of data security is even mandated by law in certain applications: in the U.S. electronic devices cannot exchange personal medical data without encrypting it first, and electronic engine controllers must not permit tampering with the data tables used to control engine emissions and performance.

Data security techniques have a reputation for being computationally intensive, mysterious, and fraught with intellectual property concerns. While some of this is true, straightforward public domain techniques that are both robust and lightweight do exist. One such technique, an algorithm called Blowfish, is perfect for use in embedded systems.

Terminology

In cryptographic circles, plaintext is the message you're trying to transmit. That message could be a medical test report, a firmware upgrade, or anything else that can be represented as a stream of bits. The process of encryption converts that plaintext message into ciphertext, and decryption converts the ciphertext back into plaintext.

Generally speaking, encryption algorithms come in two flavors, symmetric and public key. Symmetric algorithms, such as Blowfish, use the same key for encryption and decryption. Like a password, you have to keep the key secret from everyone except the sender and receiver of the message.

Public key encryption algorithms use two keys, one for encryption and another for decryption. The key used for encryption, the "public key" need not be kept secret. The sender of the message uses that public key to encrypt their message, and the recipient uses their secret decryption key, or "private key", to read it. In a sense, the public key "locks" the message, and the private key "unlocks" it: once encrypted with the public key, nobody except the holder of the private key can decrypt the message. RSA is a popular public key encryption algorithm.

Most credible encryption algorithms are published and freely available for analysis, because it's the security of the key that actually makes the algorithm secure. A good encryption algorithm is like a good bank vault: even with complete plans for the vault, the best tools, and example vaults to practice on, you won't get inside the real thing without the key.

Sometimes an encryption algorithm is restricted, meaning that the algorithm itself is kept secret. But then you can never know for sure just how weak a restricted algorithm really is, because the developer doesn't give anyone a chance to analyze it.

Encryption algorithms can be used for several kinds of data security. Sometimes you want *data integrity,* the assurance that the recipient received the same message you sent. Encryption algorithms can also provide authentication, the assurance that a message came from whom it says it came from. Some encryption algorithms can even provide nonrepudiation, a way to prove beyond a doubt (say, in a courtroom) that a particular sender was the originator of a message. And of course, most encryption algorithms can also assure data privacy, a way to prevent someone other than the intended recipient from reading the message.

Data security in practice

Let's say an embedded system wants to establish a secure data-exchange session with a laptop, perhaps over a wireless medium. At the start of the session, both the embedded system and laptop compute a private Blowfish key and public and private RSA keys. The embedded system and laptop exchange the public RSA keys and use them to encrypt and exchange their private Blowfish keys. The two machines then encrypt the remainder of their communications using Blowfish. When the communications session is over, all the keys are discarded.

In this example, it doesn't matter if someone is eavesdropping on the entire conversation. Without the private RSA keys, which never go over the airwaves, the eavesdropper can't obtain the Blowfish keys and, therefore, can't decrypt the messages passed between the two machines. This example is similar to how the OpenSSH command shell works (although OpenSSH takes additional steps to prevent the public keys from being tampered with during transit).

Now let's say that a server wants to send a firmware upgrade to a device and wants to be sure that the code isn't intercepted and modified during transit. The firmware upgrade may be delivered over a network connection, but could just as easily be delivered via a CD-ROM. In any case, the server first encrypts the firmware upgrade with its private RSA key, and then sends it to the device. The recipient decrypts the message with the server's public key, which was perhaps programmed into the device during manufacture. If the firmware upgrade is successfully decrypted, in other words a checksum of the image equals a known value, or the machine instructions look valid, the firmware upgrade is considered authentic.

The RSA algorithm is computationally expensive, although not unreasonably so for the level of functionality and security it provides. A lighter-weight approach to firmware exchange with an embedded system would be to encrypt the image with Blowfish, instead of RSA. The downside to this approach is that the Blowfish key in the embedded system has to be kept secret, which can be difficult to achieve for a truly determined attacker with hardware skills. In less extreme cases, however, Blowfish is probably fine since an attacker with such intimate knowledge of the target system and environment will likely find another way into the device anyway (in other words, simply snatching the firmware upgrade from flash memory once it's decrypted).

The Blowfish algorithm

Blowfish is a symmetric encryption algorithm, meaning that it uses the same secret key to both encrypt and decrypt messages. Blowfish is also a block cipher, meaning

MOST COMMENTED

07.14.2017

The Write Stuff: Tutorial template in high-visibility format (/electronics-blogs/the-write-stuff/4458625/The-Write-Stuff--Tutorial-template-in-high-visibility-format)

07.25.2017

A history of microprocessor debug, 1980–2016 (/electronics-blogs/say-what-/4458667/A-history-of-microprocessor-debug--1980-2016)

07.10.2017

A solution looking for a problem (/electronics-blogs/without-a-paddle/4458611/A-solution-looking-for-a-problem)

11.16.2016

Back EMF and snubber (/electronics-blogs/without-a-paddle/4443026/Back-EMF-and-snubber)

10.19.2008

Practical migration from 8-/16- to 32-bit PIC (/design/mcus-processors-and-socs/4007683/Practical-migration-from-8-16-to-32-bit-PIC)

RELATED CONTENT

10.19.2010 | SOURCE CODE

08blowfish.zip (/design/embedded/source-code/4209754/08blowfish-zip)

01.15.2014 | DESIGN

Secure Data Transaction Using Cryptographic Algorithms in ARM (/design/safety-and-security/4433977/Secure-Data-Transaction-Using-Cryptographic-Algorithms-in-ARM)

07.15.2003 | DESIGN

that it divides a message up into fixed length blocks during encryption and decryption. The block length for Blowfish is 64 bits; messages that aren't a multiple of eight bytes in size must be padded.

Blowfish is public domain, and was designed by Bruce Schneier expressly for use in performance-constrained environments such as embedded systems.[3] It has been extensively analyzed and deemed "reasonably secure" by the cryptographic community. Implementation examples are available from several sources, including the one by Paul Kocher that's excerpted in this article as Listing 1. (The complete code is available for download at ftp://ftp.embedded.com/pub/2003/08blowfish (ftp://ftp.embedded.com/pub/2003/08blowfish).)

```
/*
  Blowfish algorithm. Written 1997 by Paul Kocher (paul@cryptography.com
(mailto:paul@cryptography.com)).
  This code and the algorithm are in the0 public domain.
*/

#define MAXKEYBYTES 56    /* 448 bits */
#define N   16

typedef struct {
   uint32_t P[16 + 2];
   uint32_t S[4][256];
} BLOWFISH_CTX;

unsigned long
F(BLOWFISH_CTX *ctx, uint32_t x)
{
   uint16_t a, b, c, d;
   uint32_t y;

   d = x & 0x00FF;
   x >>= 8;
   c = x & 0x00FF;
   x >>= 8;
   b = x & 0x00FF;
   x >>= 8;
   a = x & 0x00FF;

   y = ctx->S[0][a] + ctx->S[1][b];
   y = y ^ ctx->S[2][c];
   y = y + ctx->S[3][d];

   return y;
}
void
Blowfish_Encrypt(BLOWFISH_CTX *ctx, uint32_t *xl, uint32_t *xr)
{
   uint32_t Xl;
   uint32_t Xr;
   uint32_t temp;
   int   ii;

   Xl = *xl;
   Xr = *xr;

   for (i = 0; i < n;="" ++i)="">
   {
       Xl = Xl ^ ctx->P[i];
       Xr = F(ctx, Xl) ^ Xr;

       temp = Xl;
       Xl = Xr;
       Xr = temp;
   }

   temp = Xl;
   Xl = Xr;
   Xr = temp;

   Xr = Xr ^ ctx->P[N];
```

Encrypting data with the Blowfish algorithm (/design/configurable-systems/4024599/Encrypting-data-with-the-Blowfish-algorithm)

06.28.2016 | DISCUSSION

Secure your data with self-encrypting drives (SEDs) (/electronics-blogs/max-unleashed-and-unfettered/4442291/Secure-your-data-with-self-encrypting-drives--SEDs-)

06.13.2006 | TECHNICAL PAPER

The Blowfish Encryption Algorithm (/electrical-engineers/education-training/tech-papers/4125378/The-Blowfish-Encryption-Algorithm)

```
   Xl = Xl ^ ctx->P[N + 1];


   *xl = Xl;
   *xr = Xr;
}

void
Blowfish_Decrypt(BLOWFISH_CTX *ctx, uint32_t *xl, uint32_t *xr)
{
   uint32_t Xl;
   uint32_t Xr;
   uint32_t temp;
   int   ii;

   Xl = *xl;
   Xr = *xr;

   for (i = N + 1; i > 1; --i)
   {
         Xl = Xl ^ ctx->P[i];
         Xr = F(ctx, Xl) ^ Xr;

         temp = Xl;
         Xl = Xr;
         Xr = temp;
   }

   temp = Xl;
   Xl = Xr;
   Xr = temp;

   Xr = Xr ^ ctx->P[1];
   Xl = Xl ^ ctx->P[0];

   *xl = Xl;
   *xr = Xr;
}

void
Blowfish_Init(BLOWFISH_CTX *ctx, uint16_t *key, int KeyLen)
{
   uint32_t Xl;
{
   int i, j, k;
   uint32_t data, datal, datar;

   for (i = 0; i < 4;="" i++)="">
   {
         for (j = 0; j < 256;="" j++)="" ctx-="">S[i][j] = ORIG_S[i][j];
   }

   j = 0;
   for (i = 0; i < n="" +="" 2;="" ++i)="">
   {
         data = 0x00000000;
         for (k = 0; k < 4;="" ++k)="">
         {
             data = (data < 8)="" |="">
             j = j + 1;
             if (j >= keyLen) j = 0;
         }
     ctx->P[i] = ORIG_P[i] ^ data;
   }

   datal = 0x00000000;
   datar = 0x00000000;

   for (i = 0; i < n="" +="" 2;="" i="" +="2)">
   {
         Blowfish_Encrypt(ctx, &datal, &datar);
         ctx->P[i] = datal;
```

```
        ctx->P[i + 1] = datar;
    }

    for (i = 0; i < 4;="" ++i)="">
    {
        for (j = 0; j < 256;="" j="" +="2)">
        {
            Blowfish_Encrypt(ctx, &datal, &datar);
            ctx->S[i][j] = datal;
            ctx->S[i][j + 1] = datar;
        }
    }
}

int
Blowfish_Test(BLOWFISH_CTX *ctx)
{
    uint32_t L = 1, R = 2;

    Blowfish_Init(ctx, (unsigned char*)"TESTKEY", 7);
    Blowfish_Encrypt(ctx, &L, &R);
    if (L != 0xDF333FD2L || R != 0x30A71BB4L) return (-1);

    Blowfish_Decrypt(ctx, &L, &R);
    if (L != 1 || R != 2) return (-1); return (0);
}
```

Blowfish requires about 5KB of memory. A careful implementation on a 32-bit processor can encrypt or decrypt a 64-bit message in approximately 12 clock cycles. (Not-so-careful implementations, like Kocher, don't increase that time by much.) Longer messages increase computation time in a linear fashion; for example, a 128-bit message takes about (2 x 12) clocks. Blowfish works with keys up to 448 bits in length.

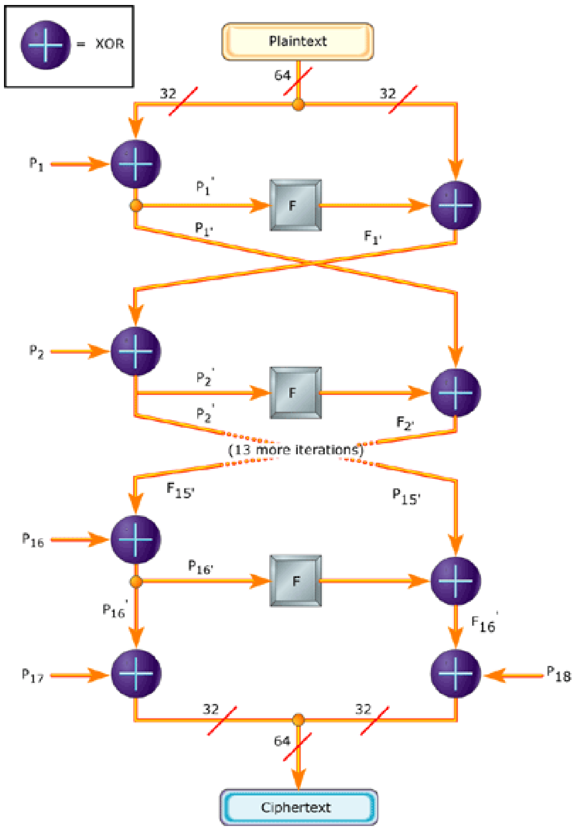

Figure 1: Blowfish algorithm

A graphical representation of the Blowfish algorithm appears in Figure 1. In this description, a 64-bit plaintext message is first divided into 32 bits. The "left" 32 bits are XORed with the first element of a P-array to create a value I'll call P', run through a transformation function called F, then XORed with the "right" 32 bits of the message to produce a new value I'll call F'. F' then replaces the "left" half of the message and P' replaces the "right" half, and the process is repeated 15 more times with successive members of the P-array. The resulting P' and F' are then XORed with the last two entries in the P-array (entries 17 and 18), and recombined to produce the 64-bit ciphertext.

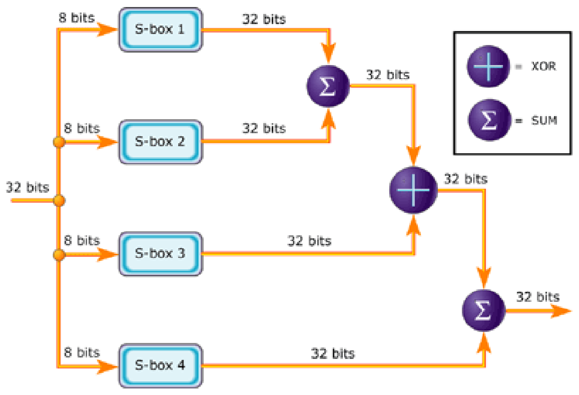

Figure 2: Graphic representation of F

A graphical representation of F appears in Figure 2. The function divides a 32-bit input into four bytes and uses those as indices into an S-array. The lookup results are then added and XORed together to produce the output.

Because Blowfish is a symmetric algorithm, the same procedure is used for decryption as well as encryption. The only difference is that the input to the encryption is plaintext; for decryption, the input is ciphertext.

The P-array and S-array values used by Blowfish are precomputed based on the user's key. In effect, the user's key is transformed into the P-array and S-array; the key itself may be discarded after the transformation. The P-array and S-array need not be recomputed (as long as the key doesn't change), but must remain secret.

I'll refer you to the source code for computing the P and S arrays and only briefly summarize the procedure as follows:

- P is an array of eighteen 32-bit integers.
- S is a two-dimensional array of 32-bit integer of dimension 4x256.
- Both arrays are initialized with constants, which happen to be the hexadecimal digits of π (a pretty decent random number source).
- The key is divided up into 32-bit blocks and XORed with the initial elements of the P and S arrays. The results are written back into the array.

- A message of all zeros is encrypted; the results of the encryption are written back to the P and S arrays. The P and S arrays are now ready for use.

Using the example code

Of course, firmware upgrades and data logs are seldom exactly 64 bits in length. To encrypt long strings of data using Blowfish, carve the message up into 64-bit blocks, encrypt each block and save the results. Pad the message with a value of your choosing to end on a 64-bit boundary. The code in the **main()** of Listing 2 does exactly this.

Listing 2: Example of Blowfish use

```
#include <stdio.h>
#include <string.h>

int
main (void)
  {
  BLOWFISH_CTX ctx;
  int n;

  /* must be less than 56 bytes */
  char *key = "a random number string would be a better key";
  int keylen = strlen(key);

  uint8_t *plaintext_string = "this is our message";
  int plaintext_len = strlen(plaintext_string);
```

```
   uint8_t ciphertext_buffer[256];
   uint8_t *ciphertext_string = &ciphertext_buffer[0];
   int ciphertext_len = 0;

   uint32_t message_left;
   uint32_t message_right;
   int block_len;

#if 1
   /* sanity test, encrypts a known message */
   n = Blowfish_Test(&ctx);
   printf("Blowfish_Test returned: %d.%s\n", n, n ? " Abort." : "");
   if (n) return n;
#endif

   Blowfish_Init(&ctx, key, keylen);

   printf("Plaintext message string is: %s\n", plaintext_string);

   /* encrypt the plaintext message string */
   printf("Encrypted message string is: ");

   while (plaintext_len)
   {
     message_left = message_right = 0UL;

   /* crack the message string into a 64-bit block (ok, really two 32-bit blocks); pad with
zeros if necessary */
     for (block_len = 0; block_len < 4;="" block_len++)="">
     {
       message_left = message_left <>
       if (plaintext_len)
       {
           message_left += *plaintext_string++;
           plaintext_len--;
       }
       else message_left += 0;
     }
     for (block_len = 0; block_len < 4;="" block_len++)="">
     {
       message_right = message_right <>
       if (plaintext_len)
       {
           message_right += *plaintext_string++;
           plaintext_len--;
       }
       else message_right += 0;
     }
   /* encrypt and print the results */
     Blowfish_Encrypt(&ctx, &message_left, &message_right);
     printf("%lx%lx", message_left, message_right);

   /* save the results for decryption below */
     *ciphertext_string++ = (uint8_t)(message_left >> 24);
     *ciphertext_string++ = (uint8_t)(message_left >> 16);
     *ciphertext_string++ = (uint8_t)(message_left >> 8);
     *ciphertext_string++ = (uint8_t)message_left;
     *ciphertext_string++ = (uint8_t)(message_right >> 24);
     *ciphertext_string++ = (uint8_t)(message_right >> 16);
     *ciphertext_string++ = (uint8_t)(message_right >> 8);
     *ciphertext_string++ = (uint8_t)message_right;
     ciphertext_len += 8;
printf("\n");

   /* reverse the process */
     printf("Decrypted message string is: ");

     ciphertext_string = &ciphertext_buffer[0];
     while(ciphertext_len)
     {
        message_left = message_right = 0UL;
        for (block_len = 0; block_len < 4;="" block_len++)="">
        {
          message_left = message_left <>
          message_left += *ciphertext_string++;
          if (ciphertext_len)
           ciphertext_len--;
        }
        for (block_len = 0; block_len < 4;="" block_len++)="">
        {
           message_right = message_right <>
           message_right += *ciphertext_string++;
           if (ciphertext_len)
           ciphertext_len--;
        }

        Blowfish_Decrypt(&ctx, &message_left, &message_right);

    /* if plaintext message string padded, extra zeros here */

        printf("%c%c%c%c%c%c%c%c",
        (int)(message_left >> 24), (int)(message_left >> 16),
        (int)(message_left >> 8), (int)(message_left),
        (int)(message_right >> 24), (int)(message_right >> 16),
        (int)(message_right >> 8), (int)(message_right));
}

printf("\n");

return 0;
}
```

Now is a good time to start thinking about adding data integrity and privacy capabilities to your embedded system. The Blowfish algorithm is an excellent choice for encryption, since it's lightweight, public domain, and considered secure even after extensive analysis.

Bill Gatliff is a consultant who specializes in solving embedded development problems using free software tools. He's the creator of the gdbstubs library, a free collection of embeddable stubs for the GNU debugger. You can reach him at bgat@billgatliff.com (mailto:bgat@billgatliff.com).

End notes:

1. Not an actual log per se, but so-called ephemerides information that allows the device to find GPS transmitters without doing a time-consuming search of the entire GPS spectrum. Such information can also be used to pinpoint the receiver's location at a previous point in time. Because of this capability, GPS receivers are routinely collected and analyzed during searches by law enforcement. A digital signature would authenticate the ephimeride, verifying that it hadn't been tampered with or rendered invalid before being used as evidence.
   Back
2. In the U.S., commercial automotive systems do this to prevent warranty claims for user-damaged hardware; in Europe, it's to prevent speeding.
   Back
3. Schneier, Bruce. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition.* New York, NY: John Wiley & Sons, 1995.
   Back

WRITE A COMMENT

Subscribe to RSS updates all articles (/rss/all) or category ▾

DEVELOPMENT CENTERS

All Articles (/development)

Configurable Systems (/development/configurable-systems)

Connectivity (/development/connectivity)

Debug & Optimization (/development/debug-and-optimization)

MCUs, Processors & SoCs (/development/mcus-processors-and-socs)

Operating Systems (/development/operating-systems)

Power Optimization (/development/power-optimization)

Programming Languages & Tools (/development/programming-languages-and-tools)

Prototyping & Development (/development/prototyping-and-development)

Real-time & Performance (/development/real-time-and-performance)

Real-world Applications (/development/real-world-applications)

Safety & Security (/development/safety-and-security)

System Integration (/development/system-integration)

ESSENTIALS & EDUCATION

Products (/products/all)

News (/news/all)

Source Code Library (/education-training/source-codes)

Webinars (/education-training/webinars)

Courses (/education-training/courses)

Tech Papers (/education-training/tech-papers)

COMMUNITY

Insights (/insights)

Forums (http://forums.embedded.com)

Events (/events)

ARCHIVES

Embedded Systems Programming / Embedded Systems Design Magazine (/magazines)

Newsletters (/archive/Embedded-com-Tech-Focus-Newsletter-Archive)

Videos (/videos)

Collections (/collections/all)

ABOUT US

About Embedded (/aboutus)

Contact Us (/contactus)

Newsletters (/newsletters)

Advertising (/advertising)

Editorial Contributions (/editorial-contributions)

Site Map (/site-map)

ASPENCORE NETWORK